Quickstrom Documentation

Release 0.1.0

Oskar Wickström

Nov 04, 2022

CONTENTS

Quickstrom is a new autonomous testing tool for the web. It can find problems in any type of web application that renders to the DOM. Quickstrom automatically explores your application and presents minimal failing examples. Focus your effort on understanding and specifying your system, and Quickstrom can test it for you.

Interested? Let's get started!

CHAPTER

ONE

DOCUMENTATION

If you're new to Quickstrom, start here:

- *Installation*: how to get Quickstrom running on your computer
- *Writing Your First Specification*: the entry-level tutorial

The documentation is split up into sections depending on the type of document:

- *Topics*: high-level explanations of concepts and how they fit together
- *Tutorials*: step-by-step guides focused on learning
- *How-To Guides*: short guides to achieve specific goals

CHAPTER

TWO

STAYING UPDATED

Sign up for the the newsletter.

2.1 Installation

Follow these steps to install Quickstrom locally. These are the currently supported installation methods:

2.1.1 Installing with Nix

Follow these steps to install Quickstrom using Nix.

Prerequisites

- Nix (see nix.dev for installation instructions and guides)

Installing with Nix

To install the `quickstrom` executable, use Cachix and Nix to get the executable:

```
$ cachix use quickstrom
$ nix-env -iA quickstrom -f https://github.com/quickstrom/quickstrom/tarball/main
```

Note: If the installation fails with "too many open files", see How do I set the ulimit in a nix build shell?.

Verify that Quickstrom is now available in your environment:

```
$ quickstrom version
```

You need to run a WebDriver server for Quickstrom checks to work. This user documentation mostly uses GeckoDriver and Firefox, but you can use other browsers and WebDriver servers.

Install GeckoDriver using Nix:

```
$ nix-env -i geckodriver
```

You're now ready to *check webapps using Quickstrom*.

2.1.2 Installing with Docker

QuickStrom provides a Docker image as an easy installation method. Download the image using Docker:

```
$ docker pull quickstrom/quickstrom:latest
```

Verify that Quickstrom can now be run using Docker:

```
$ docker run quickstrom/quickstrom:latest quickstrom version
```

Installing a WebDriver Server

A WebDriver server must be running and available on `127.0.0.0:4444` for Quickstrom to work. In this example we'll use Geckodriver and Firefox. Download a Geckodriver image using Docker:

```
$ docker pull instrumentisto/geckodriver
```

You can now run Geckodriver and the `quickstrom` executable with `docker run`:

```
$ docker run -d -p 4444:4444 instrumentisto/geckodriver
$ docker run \
  --network=host \
  --mount=type=bind,source=$PWD/specs,target=/specs \
  quickstrom/quickstrom:latest \
  quickstrom check \
  /specs/Example.spec.purs \
  https://example.com
```

There's a lot of things going in the above session. Let's look at what each line does:

1. Launch a geckodriver instance in a separate *detached* container
2. Uses *docker run* to execute a program inside the container
3. Uses host network to get easy access to Geckodriver (see below)
4. Mounts a host directory containing specification(s) to `/specs` in the container filesystem
5. Uses the image `quickstrom/quickstrom` with the `latest` target
6. Runs `quickstrom` with the `check` command
7. Passes a path to a specification file in the mounted directory
8. Passes an origin URI (this could also be a file path in the mounted directory)

There are other ways of setting up network access between Docker containers. Using host networking is convenient in this case, but you might require or prefer another method.

2.1.3 Accessing a Server on the Host

If you wish to run Quickstrom in Docker and test a website being hosted by the Docker host system you can set the url to `localhost` (or `host.docker.internal` for MacOS).

```
$ docker run \
  --network=host \
  --mount=type=bind,source=$PWD/specs,target=/specs \
  quickstrom/quickstrom:latest \
  quickstrom check \
  /specs/Example.spec.purs \
  http://localhost:3000 # or http://host.docker.internal:3000 for MacOS (You may have to␣
↪disable HOST checking if you get "Invalid Host header" messages)
```

2.2 Topics

This section explains how the different parts of Quickstrom work and fit together at a high level.

2.2.1 How It Works

In Quickstrom, a tester writes *specifications* for web applications. When *checking* a specification, the following happens:

1. Quickstrom navigates to the *origin page*, and awaits the *readyWhen* condition, that a specified element is present in the DOM.
2. It generates a random sequence of actions to simulate user interaction. Many types of actions can be generated, e.g. clicks, key presses, focus changes, reloads, navigations.
3. Before each new action is picked, the DOM state is checked to find only the actions that are possible to take. For instance, you cannot click buttons that are not visible. From that subset, Quickstrom picks the next action to take.
4. After each action has been taken, Quickstrom queries and records the state of relevant DOM elements. The sequence of actions takens and observed states is called a *behavior*.
5. The specification defines a proposition, a logical formula that evaluates to *true* or *false*, which is used to determine if the behavior is *accepted* or *rejected*.
6. When a rejected behavior is found, Quickstrom *shrinks* the sequence of actions to the *smallest*, *still failing*, behavior. The tester is presented with a minimal failing test case based on the original larger behavior.

Now, how do you write specifications and propositions? Let's have a look at *The Specification Language*.

2.2.2 The Specification Language

In Quickstrom, the behavior of a web application is described in a specification language. It's a propositional temporal logic and functional language, heavily inspired by TLA+ and LTL, most notably adding web-specific operators. The specification language of Quickstrom is based on PureScript.

Like in TLA+, specifications in Quickstrom are based on state machines. A *behavior* is a finite sequence of states. A *step* is a tuple of two successive states in a behavior. A specification describes valid *behaviors* of a web application in terms of valid states and transitions between states.

As in regular PureScript, every expression evaluates to a *value*. A *proposition* is a boolean expression in a specification, evaluating to either `true` or `false`. A specification that accepts *any* behavior could therefore be:

```
module Spec where

proposition = true

... -- more definitions, explained further down
```

To define a useful specification, though, we need to perform *queries* and describe how things change over time (using *temporal operators*).

Queries

Quickstrom provides two ways of querying the DOM in your specification:

- `queryAll`
- `queryOne`

Both take a CSS selector and a record of element state specifiers, e.g. attributes or properties that you're interested in.

For example, the following query finds all buttons, including their text contents and disabled flags:

```
myButtons = queryAll "button" { textContent, disabled }
```

The type of the above expression is:

```
Array { textContent :: String, disabled :: Boolean }
```

You can use regular PureScript function to map, filter, or whatever you'd like, on the array of button records.

In contrast to `queryAll` returning an `Array`, `queryOne` returns a `Maybe`.

Temporal Operators

In Quickstrom specifications, there are three core temporal operators:

- `next :: forall a. a -> a`
- `always :: Boolean -> Boolean`
- `until :: Boolean -> Boolean -> Boolean`

They change the *modality* of the sub-expression, i.e. in what state of the recorded behavior it is evaluated.

There are also utility functions built on top of the temporal operators:

- `unchanged :: Eq a => a -> Boolean`

Let's go through the operators and utility functions provided by Quickstrom!

Always

Let's say we have the following proposition:

```
proposition = always (title == Just "Home")

title = map _.textContent (queryOne "h1" { textContent })
```

In every observed state the sub-expression must evaluate to `true` for the proposition to be true. In this case, the text content of the `h1` must always be "Home".

Until

Until takes two parmeters: the prerequisite condition and the final condition. The prerequisite must hold `true` in all states until the final condition is `true`.

```
proposition = until (loading == Just "loading...") (title == Just "Home")

loading = map _.textContent (queryOne "loading" { textContent })
title = map _.textContent (queryOne "h1" { textContent })
```

In this case, we presumably load the "Home" text from somewhere else, so we wait until the loading is done, and then assert that the title must be set accordingly.

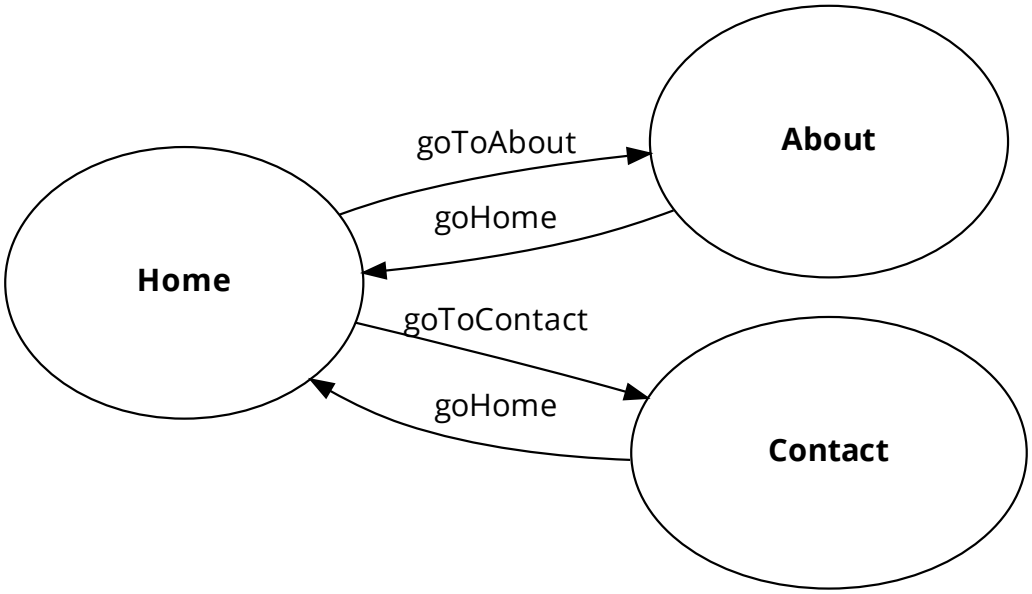
Next

Let's modify the previous proposition to describe a state change:

```
proposition = always (goToAbout || goToContact || goHome)

goToAbout = title == Just "Home" && next title == Just "About"

goToContact = title == Just "Home" && next title == Just "Contact"

goHome = title /= Just "Home" && next title == Just "Home"

title = map _.textContent (queryOne "h1" { textContent })
```

We're now saying that it's always the case that one or another state transition occurs. An state transition is represented as a boolean expression, using queries and `next` to describe the current and the next state.

The `goToAbout`, `goToContact`, and `goHome` transitions specify how the title of the page changes, and the `proposition` thus describes the system as a state machine. It can be visualized as follows:



Unchanged

In addition to the core temporal operators, the `unchanged` operator is a utility for stating that something does *not* change:

```
unchanged :: forall a. Eq a => a -> Boolean
unchanged x = x == next x
```

It's useful when expressing state transitions, specifying that a certain queried value should be the same both before and after a particular transition.

For instance, let's say we have a bunch of top-level definitions, all based on DOM queries, describing a user profile:

```
userName :: String
userName = ...

userProfileUrl :: String
userProfileUrl = ...
```

We can say the user profile information should not change in a transition `t` by passing an array of those values:

```
t = unchanged [userName, userProfileUrl]
    && ... -- actual changes in transition
```

Actions

We must instruct Quickstrom what actions it should try. The `actions` definition in a specification module is where you list possible actions.

```
actions :: Actions
actions = [ action1, action2, ... ]
```

It's an array of values, where each value describes an action or a fixed sequence of actions. Each action also carries a weight, which specifies the intended probability of the action being picked, relative to the other actions.

The default weight is `1`. To override it, use the `weighted` function:

```
click "#important-action" `weighted` 10
```

To illustrate, in the following array of actions, the probability of `a1` being picked is 40%, while the others are at 20% each. This is assuming the action (or the first action in each sequence) is *possible* at each point a sequence is being picked.

```
actions = [
    a1 `weighted` 2,
    a2,
    a3,
    a4
  ]
```

Action Sequences

An action sequence is either a single action or a fixed sequence of actions. Here's a simple sequence:

```
backAndForth = click "#back" `followedBy` click "#forward"
```

A sequence of actions is always performed in its entirety when picked, as long as the first action in the sequence is considered possible by the test runner.

Actions

The available actions are provided in the Quickstrom library:

- `focus`
- `keyPress`
- `enterText`
- `click`
- `clear`
- `await`
- `awaitWithTimeoutSecs`
- `navigate`
- `refresh`

Along with those functions, there are some aliases for common actions. For instance, here's the definition of `foci`:

```
-- | Generate focus actions on common focusable elements.
foci :: Actions
foci =
   [ focus "input"
   , focus "textarea"
   ]
```

More actions and aliases should be introduced as Quickstrom evolves.

Example

As an example of composing actions and sequences of actions, here's a collection of actions that try to log in or to click a buy button:

```
actions =
   [ focus "input[type=password]"
       `followedBy` enterText "$ecr3tz"
       `followedBy` click "input[type=submit][name=log-in]"
   , click "input[type=submit][name=buy]"
   ]
```

Note: When specifying complex web applications, one must often carefully pick selectors, actions, and weights, to effectively test enough within a reasonable time. Aliases like `clicks` and `foci` might not work well in such situations.

2.2.3 Checking

To check a web application against a specification, use the `quickstrom check` command. Supply the path to the specification file along with the origin URL.

```
$ quickstrom check \
   /path/to/my/specification \
   http://example.com
```

The origin can also be a local file:

```
$ quickstrom check \
   /path/to/my/specification \
   /path/to/my/webapp.html
```

Note: To check a specification, you must have a running WebDriver server. Most guides in this user documentation use GeckoDriver and Firefox. Other options are discussed below.

Cross-Browser Testing

Quickstrom currently supports these browsers:

- Firefox (`firefox`)
- Chrome/Chromium (`chrome`)

Unless specified, the default browser used is Firefox. To override, use the `--browser` option and set the appropriate browser when running the `check` command:

```
$ quickstrom check \
   --browser=chrome \
   ... # more options
```

If you need to specify the executable, use `--browser-binary`:

```
$ quickstrom check \
   --browser=chrome \
   --browser-binary=/path/to/google-chrome \
   ... # more options
```

WebDriver Options

If your WebDriver server is running on a different host, port, or path than the default (`http://127.0.0.1:4444`), you can override those options:

```
$ quickstrom check \
   --webdriver-host=hub.example.com \
   --webdriver-port=12345 \
   --webdriver-path="/wd/hub" \
   ... # more options
```

2.2.4 Trailing State Changes

By default, Quickstrom only listens for a single DOM state change after each action it performs. This behavior can be overriden, so that it waits for a configurable number of *trailing state changes*.

The term *trailing* refers primarily to asynchronous changes that occur as result of an action. For example:

- the user agent clicks a button
- a loading indicator is shown immediately
- an HTTP request is performed
- later, the result of the request is printed

In this example, the loading indicator being shown is the first state change. The result of the HTTP request being shown is the trailing state change.

Some systems change the state of the DOM without any dependence on user action, and do so infinitely. For instance, a clock (hopefully) keeps ticking, no matter what the user is up to. It doesn't make much sense to think of a clock's behavior as "trailing". However, it's still possible to test a finite subsequence of such a behavior using Quickstrom and trailing state changes.

Command-Line Options

The command-line options available are:

- `--max-trailing-state-changes=NUMBER`: how many trailing state changes Quickstrom will try to observe.
- `--trailing-state-change-timeout=MILLISECONDS`: maximum time that it will wait for a change of DOM state. The timeout doubles for every subsequent trailing state change that is awaited.

Let's say we set the following options:

```
--max-trailing-state-changes=3
--trailing-state-change-timeout=200
```

Then the DOM state changes would be observed at the following times:

1. initial state, immediately
2. first trailing state, at most 200ms after #1
3. second trailing state, at most 400ms after #2
4. third trailing state, at most 800ms after #3

It's *at most*, because Quickstrom observes the DOM and can pick up state changes as they happen.

2.2.5 Reporters

After a Quickstrom check completes, one or more *reporters* run. They report the result of the check in different formats. The following reporters are available:

- `console`
- `html`
- `json`

Invoke reporters by passing the `--reporter=<NAME>` option to the `check` command.

Console

The console reporter is invoked by default. It prints a trace and summary to the console when a check fails. The trace contains information about the state of queried elements at each state in the behavior, along with the actions taken by Quickstrom.

HTML

The HTML reporter creates a report for web browsers in a given directory. The report is an interactive troubleshooting tool based on state transitions, showing screenshots and overlayed state information for the queried elements.

To set the directory to generate the report in, use the option `--html-report-directory=<DIR>`.

The HTML report directory must be served through an HTTP server in order to avoid problems with CORS. If you have Python 3 installed, serve it with the following command:

```
$ python3 -m http.server -d <DIR>
```

JSON

The JSON report works similarly to the HTML report, except it generates only a JSON file and screenshots, no HTML files. In the report directory you'll find a file `report.json` that you can work with.

To set the directory to generate the report in, use the option `--json-report-directory=<DIR>`.

2.2.6 Troubleshooting

This documents collects some common problems and tips on how to identify what's not working correctly.

If you're troubleshooting a failing Quickstrom check, make sure to enable debug logs:

```
$ quickstrom check --log-level=DEBUG ...
```

No WebDriver Session

If you get the following error when using GeckoDriver (especially after having successfully run before):

```
quickstrom: user error (E NoSession)
```

It's probably because the WebDriver package in Quickstrom failed to clean up its session. This is a known bug. To work around it, restart Geckodriver and rerun your Quickstrom command.

2.3 Tutorials

The following tutorials are detailed end-to-end guides to help you learn how to use Quickstrom.

2.3.1 Writing Your First Specification

In this tutorial we'll specify and check an *audio player* web application using the free version of Quickstrom.

The tutorial assumes you're running on a Unix-like operating system and that you have Docker installed. You may run this using *other installation methods* for Quickstrom and your WebDriver server, but all commands in this document are using Docker.

Open up a terminal and create a new directory to work in:

```
$ mkdir my-first-spec
$ cd my-first-spec
```

Installing with Docker

In this tutorial you need a working installation of Docker. Head over to docker.com and set it up if you haven't already.

Next, pull the QuickStrom image using Docker:

```
$ docker pull quickstrom/quickstrom:latest
```

Finally, we need a WebDriver server. Pull that down with Docker, too:

```
$ docker pull selenium/standalone-chrome:3.141.59-20200826
```

Downloading the Audio Player

The web application we're going to test is already written. Download it using `curl`:

```
$ curl -L https://github.com/quickstrom/quickstrom/raw/main/examples/AudioPlayer.html -o␣
↪AudioPlayer.html
```

If you don't have `curl` installed, you can download it from this URL using your web browser. Make sure you've saved it our working directory as `AudioPlayer.html`.

```
$ ls
AudioPlayer.html
```

OK! We're now ready to write our specification.

A Minimal Specification

We'll begin by writing a specification that always makes the tests pass. Create a new file `AudioPlayer.spec.purs` and open it in your text editor of choice:

```
$ touch AudioPlayer.spec.purs
$ $EDITOR AudioPlayer.spec.purs
```

Type in the following in the file and save it:

```
1  module AudioPlayer where
2
3  import Quickstrom
4  import Data.Maybe (Maybe(..))
5
6  readyWhen :: Selector
7  readyWhen = ".audio-player"
8
9  actions :: Actions
10 actions = clicks
11
12 proposition :: Boolean
13 proposition = true
```

A bunch of things are going on in this specification. Let's break it down line by line:

- **Line 1:** We declare the `AudioPlayer` module. We must have a module declaration, but it can be named whatever we like.
- **Line 3-4:** We import the Quickstrom module. This is where we find definitions for DOM queries, actions, and logic. We also import *Maybe* which we'll need later on.
- **Line 6-7:** The `readyWhen` definitions tells Quickstrom to wait until there's an element in the DOM that matches this CSS selector. After this condition holds, Quickstrom will start performing actions. We use `.audio-player` as the selector, which is used as a class for the top-level `div` in the audio player web application.

- **Line 9-10:** Our `actions` specify what Quickstrom should try to do. In this case, we want it to click any available links, buttons, and so on.
- **Line 12-13:** In the `proposition`, we specify what it means for the system under test to be valid. For now, we'll set it to `true`, meaning that *any* behavior is considered valid.

Running Tests

Let's run some tests!

First, we need a Docker network. Let's name it `quickstrom`:

```
$ docker network create quickstrom
```

Next, from within your `my-first-spec` directory, launch a ChromeDriver instance in the background:

```
$ docker run --rm -d \
    --network quickstrom \
    --name webdriver \
    -v /dev/shm:/dev/shm \
    -v $PWD:/my-first-spec \
    selenium/standalone-chrome:3.141.59-20200826
```

Notice how we mount the current working directory to `/my-first-spec` in the container. We do this to let Chrome access the `AudioPlayer.html` file.

Now, let's launch Quickstrom, again from within your `my-first-spec` directory:

```
$ docker run --rm \
  --network quickstrom \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom \
  quickstrom check \
  --webdriver-host=webdriver \
  --webdriver-path=/wd/hub \
  --browser=chrome \
  --tests=5 \
  /my-first-spec/AudioPlayer.spec.purs \
  /my-first-spec/AudioPlayer.html
```

After some time, you should see an output like the following:

```
Running 5 tests...

---------------------------

20 Actions
Test passed!

---------------------------

40 Actions
Test passed!

---------------------------
```

(continues on next page)

(continued from previous page)

```
60 Actions
Test passed!

---------------------------

80 Actions
Test passed!

---------------------------

100 Actions
Test passed!

---------------------------


Passed 5 tests.
```

Cool, we have it running! So far, though, we haven't done much testing. Quickstrom is happily clicking its way around the web application, but whatever it finds we say "it's all good!" Let's make our specification actually say something about the audio player's intended behavior.

Refining the Proposition

Our system under test (`AudioPlayer.html`) is very simple. There's a button for playing or pausing the audio player, and there's a time display.

Our specification will describe how the player should work. Informally, we state the requirements as follows:

- Initially, the player should be `paused`
- When `paused`, and when the play/pause button is clicked, it should transition to the `playing` state
- When in the `playing` state, the time display should reflect the progress with a ticking minutes and seconds display
- When `playing`, and when the play/pause button is clicked, it should go to the `paused` state
- In the `paused` state, the button should say "Play"
- In the `playing` state, the button should say "Pause"

Let's translate those requirements to a formal specification in Quickstrom.

Begin by defining two helpers, extracting the text content of the time display and the play/pause button. Place these definitions at the bottom of `AudioPlayer.spec.purs`:

```
timeDisplayText :: Maybe String
timeDisplayText =
  map _.textContent (queryOne ".time-display" { textContent })

buttonText :: Maybe String
buttonText =
  map _.textContent (queryOne ".play-pause" { textContent })
```

Next, we'll change the `proposition`. Remove `true` and type in the following code:

```
proposition :: Boolean
proposition =
  let
    playing = ?playing

    paused = ?paused

    play = ?play

    pause = ?pause

    tick = ?tick
  in
    paused && always (play || pause || tick)
```

All those terms prefixed with question marks are called *holes*. A hole is a part of a program that is yet to be written, like a placeholder. We'll fill the holes one by one.

The last line in our proposition can be read in English as:

> Initially, the record player is paused. From that point, one can either play or pause, or the time can tick while playing, all indefinitely.

OK, onto filling the holes!

Filling Holes in the Specification

Let's start with the definitions that describe *states* that the program can be in.

The `playing` definition should describe what it means to be in the `playing` state. We specify it by stating that the button text should be "Pause". Replace `?playing` with the following expression:

```
buttonText == Just "Pause"
```

The `Just "Pause"` means that there is a matching element with text content "Pause". `Nothing` would mean that the query didn't find any element.

Similary, the `paused` state is defined as the button text being "Play". Replace `?paused` with:

```
buttonText == Just "Play"
```

We've now specified the two states that the audio player can be in. Next, we specify *transitions* between states.

The definition `play` describes a transition between `paused` and `playing`. Replace the hole `?play` with the following expression:

```
paused && next playing
```

OK, so what's going on here? We specify that the current state is `paused`, and that the next state is `playing`. That's how we encode state transitions.

The `pause` transition is similar. Replace `?pause` with the following expression:

```
playing && next paused
```

Finally, we have the `tick`. When we're in the `playing` state, the time display changes its text on a `tick`. The displayed time should be monotonically increasing, so we compare alphabetically the current and the next time.

Replace the hole `?tick` with the following expression:

```
playing
  && next playing
  && timeDisplayText < next timeDisplayText
```

If the time display would go past "99:59", we'd get into trouble with this specification. But because we won't run tests for that long, we can get away with the string comparison.

That's it! We've filled all the holes. Your proposition should now look something like this:

```
proposition :: Boolean
proposition =
  let
    playing = buttonText == Just "Pause"

    paused = buttonText == Just "Play"

    play = paused && next playing

    pause = playing && next paused

    tick =
      playing
        && next playing
        && timeDisplayText < next timeDisplayText
  in
    paused && always (play || pause || tick)
```

Let's run some more tests.

Catching a Bug

Run Quickstrom again, now that we've fleshed out the specification:

```
$ docker run --rm \
  --network quickstrom \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom \
  quickstrom check \
  --webdriver-host=webdriver \
  --webdriver-path=/wd/hub \
  --browser=chrome \
  --tests=5 \
  /my-first-spec/AudioPlayer.spec.purs \
  /my-first-spec/AudioPlayer.html
```

You'll see a bunch of output, involving shrinking tests and more. It should end with something like the following:

```
1. State
  • .play-pause
      -
          - property "textContent" = "Play"
  • .time-display
```

(continues on next page)

(continued from previous page)

```
      -
        - property "textContent" = "00:00"
2. click button[0]
3. click button[0]
4. State
  • .play-pause
      -
        - property "textContent" = "Play"
  • .time-display
      -
        - property "textContent" = "NaN:NaN"

Failed after 1 tests and 4 levels of shrinking.
```

Whoops, look at that! It says that the time display shows "NaN:NaN". We've found our first bug using Quickstrom!

Open up `AudioPlayer.html`, and change the following lines near the end of the file:

```
case "pause":
    return await inPaused();
```

They should be:

```
case "pause":
    return await inPaused(time); // <-- this is where we must pass in time
```

Rerun the tests using the same `quickstrom` command as before. All tests pass!

Are we done? Is the audio player correct? Not quite.

Transitions Based on Time

The audio player transitions between states mainly as a result of user action, but not only. A `tick` transition (going from `playing` to `playing` with an incremented progress) is triggered by *time*.

We'll try tweaking Quickstrom's options related to *trailing state changes* to test more of the time-related behavior of the application.

Run new tests by executing the following command:

```
$ docker run --rm \
  --network quickstrom \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom \
  quickstrom check \
  --webdriver-host=webdriver \
  --webdriver-path=/wd/hub \
  --browser=chrome \
  --tests=5 \
  --max-trailing-state-changes=1 \
  --trailing-state-change-timeout=500 \
  /my-first-spec/AudioPlayer.spec.purs \
  /my-first-spec/AudioPlayer.html
```

You should see output such as the following:

```
1. State
  • .play-pause
      -
          - property "textContent" = "Play"
  • .time-display
      -
          - property "textContent" = "00:00"
2. click button[0]
3. State
  • .play-pause
      -
          - property "textContent" = "Play"
  • .time-display
      -
          - property "textContent" = "00:01"

Failed after 1 tests and 5 levels of shrinking.
```

Look, another bug! It seems that there are `tick` transitions even though the play/pause button indicates that we're in the `paused` state.

In fact, the problem is the button text, not the time display. I'll leave it up to you to find the error in the code, fix it, and make the tests pass.

Summary

Congratulations! You've completed the tutorial, created your first specification, and found multiple bugs.

Have we found all bugs? Possibly not. This is the thing with testing. We can't know if we've found all problems. However, Quickstrom tries very hard to find more of them for you, requiring less effort.

This tutorial is intentionally fast-paced and low on theory. Now that you've got your hands dirty, it's a good time to check out *The Specification Language* to learn more about the operators in Quickstrom.

2.4 How-To Guides

Achieve specific goals with these cookbook-style guides. They're not as thorough as the *Tutorials* and normally expect you to know how to set up and run Quickstrom already.

2.4.1 Broken Links

This is simple specification used for finding broken internal links. External links are not followed. It's based on Domen Kožar's gist.[1]

```
module BrokenLinks where

import Quickstrom
import Data.Foldable (any)
import Data.Maybe (maybe)
```

(continues on next page)

[1] Domen Kožar wrote the original specification for Cachix and published it along with a GitHub Actions setup: https://gist.github.com/domenkozar/71135bf7aa6d50d6911fb74f4dcb4bad

(continued from previous page)

```
import Data.Tuple (Tuple(..))
import Data.String.CodeUnits (contains)
import Data.String.Pattern (Pattern(..))

readyWhen = "body"

actions :: Actions
actions = [
    -- Only links beginning with a slash are followed. We
    -- could also use an absolute base URL, e.g:
    --
    --     click "a[href^='https://example.com']"
    --
    click "a[href^='/']"
]

-- We're interested in finding links that lead to pages
-- rendered with these status codes in the heading.
patterns = map Pattern [ "404", "500" ]

-- In our system's error pages, status codes are rendered
-- in an <h1> element.
heading =
    maybe "" _.textContent (queryOne "h1" { textContent })

-- Check if the heading has any of the error codes in the
-- text.
hasErrorCode =
    any (\p -> contains p heading) patterns

-- This is the safety property. At no point, following
-- only internal links, should we see error codes.
proposition = always (not hasErrorCode)
```

Tweak the patterns and the precicate to fit your use case. You might not have status codes rendered, but texts like "Page not found".

2.4.2 Testing in GitHub Actions

Quickstrom can be run in a continuous integration (CI) workflow to find problems early. Here's a configuration for GitHub Actions that checks a website on every commit to the `main` branch. It's based on Domen Kožar's gist.[1]

```
name: "Quickstrom integration tests"
on:
  push:
    branches:
      - main
jobs:
  build:
```

(continues on next page)

[1] Domen Kožar wrote the original GitHub Action configuration for Cachix: https://gist.github.com/domenkozar/71135bf7aa6d50d6911fb74f4dcb4bad

(continued from previous page)

```
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2.3.4

    # We use `install-nix-action` and `cachix-action` to quickly install the
    # latest Quickstrom from a binary cache.
    - uses: cachix/install-nix-action@v12
    - uses: cachix/cachix-action@v8
      with:
        name: quickstrom
    - run: nix-env -iA quickstrom -f https://github.com/quickstrom/quickstrom/tarball/
↪main

    # We install and run Geckodriver in the background, so that we can run
    # tests using Firefox.
    - run: nix-env -i geckodriver -f https://github.com/NixOS/nixpkgs/tarball/nixos-21.05
    - run: geckodriver&

    # Finally, run tests! This assumes there's a file called
    # `example.spec.purs` in the root of the GitHub repository.
    - run: quickstrom check example.spec.purs https://example.com
```

Replace the placeholder paths and URLs.

Next Steps

- You might want to run tests in Chrome instead. See *Checking* for instructions on using other browsers.
- If you'd like to check multiple specs and in multiple browsers, see matrix configurations in the GitHub Actions documentation.

2.5 FAQ

Here are some frequently asked questions about Quickstrom:

2.5.1 Isn't this just property-based testing for web applications?

Well, not exactly. Quickstrom definitely is a form of property-based testing (PBT), but it's not *only* that. Being specifically designed for testing web applications, Quickstrom can reduce the amount of work you need to do in order to test properties of your system:

- Quickstrom discovers and performs actions automatically
- You specify only the properties you care about, and you don't have to write a fully functional model
- Quickstrom aims to (in the future) perform fault injection automatically, such as delaying, cancelling, or manipulating XHR responses, run in concurrent tabs, manipulate cookies or web storage, etc

It might be useful to think of Quickstrom as a mix of PBT, black-box browser testing, and a specification system like TLA+. One aim is "to be the Jepsen for web applications."

2.5.2 Why should I use Quickstrom instead of a model-based property test?

You might argue that this is just property-based testing, and that you could do this with state machine testing. And you'd be right! Similar tests could be written using a state machine model, WebDriver, and property-based testing.

With Quickstrom, however, you don't have to write a model that fully specifies the behavior of your system. Instead, you describe the most important state transitions and leave the rest unspecified. You can gradually adopt Quickstrom and improve your specifications over time.

Furthermore, in problem domains where there's lots of of *essential complexity*, models tend to become as complex. For example, it's often hard to find a naive implementation for your model when your modelling a business system with a myriad of arbitrary rules.